MEM02-C. Immediately cast the result of a memory allocation function call into a pointer to the allocated type

An object of type `void *` is a generic data pointer. It can point to any data object. For any incomplete or object type `T`, C permits implicit conversion from `T *` to `void *` or from `void *` to `T *`. C Standard memory allocation functions `aligned_alloc()`, `malloc()`, `calloc()`, and `realloc()` use `void *` to declare parameters and return types of functions designed to work for objects of different types.

For example, the C library declares `malloc()` as

```
void *malloc(size_t);
```

Calling `malloc(s)` allocates memory for an object whose size is `s` and returns either a null pointer or a pointer to the allocated memory. A program can implicitly convert the pointer that `malloc()` returns into a different pointer type.

Because objects returned by the C Standard memory allocation functions are implicitly converted into any object type, we recommend casting the results of these functions into a pointer of the allocated type because it increases the chances that the compiler will catch and diagnose a mismatch between the intended type of the object and the actual type of the object.

Noncompliant Code Example

The argument to `malloc()` can be *any* value of (unsigned) type `size_t`. If the program uses the allocated storage to represent an object (possibly an array) whose size is greater than the requested size, the behavior is undefined. The implicit pointer conversion lets this slip by without complaint from the compiler.

Consider the following example:

```
#include <stdlib.h>

typedef struct gadget gadget;
struct gadget {
  int i;
  double d;
};

typedef struct widget widget;
struct widget {
  char c[10];
  int i;
  double d;
};

widget *p;

/* ... */

p = malloc(sizeof(gadget)); /* Imminent problem */
if (p != NULL) {
  p->i = 0;                 /* Undefined behavior */
  p->d = 0.0;               /* Undefined behavior */
}
```

An implementation may add padding to a gadget or widget so that `sizeof(gadget)` equals `sizeof(widget)`, but this is highly unlikely. More likely, `sizeof(gadget)` is less than `sizeof(widget)`. In that case,

```
p = malloc(sizeof(gadget)); /* Imminent problem */
```

quietly assigns `p` to point to storage too small for a widget. The subsequent assignments to `p->i` and `p->d` will most likely produce memory overruns.

Casting the result of `malloc()` to the appropriate pointer type enables the compiler to catch subsequent inadvertent pointer conversions. When allocating individual objects, the "appropriate pointer type" is a pointer to the type argument in the `sizeof` expression passed to `malloc()`.

In this code example, `malloc()` allocates space for a `gadget`, and the cast immediately converts the returned pointer to a `gadget *`:

```
widget *p;

/* ... */

p = (gadget *)malloc(sizeof(gadget)); /* Invalid assignment */
```

This lets the compiler detect the invalid assignment because it attempts to convert a `gadget *` into a `widget *`.

Compliant Solution (Hand Coded)

This compliant solution repeats the same type in the `sizeof` expression and the pointer cast:

```
widget *p;

/* ... */

p = (widget *)malloc(sizeof(widget));
```

Compliant Solution (Macros)

Repeating the same type in the `sizeof` expression and the pointer cast is easy to do but still invites errors. Packaging the repetition in a macro, such as

```
#define MALLOC(type) ((type *)malloc(sizeof(type)))
```

further reduces the possibility of error.

```
widget *p;

/* ... */

p = MALLOC(widget);   /* OK */
if (p != NULL) {
  p->i = 0;           /* OK */
  p->d = 0.0;         /* OK */
}
```

Here, the entire allocation expression (to the right of the assignment operator) allocates storage for a `widget` and returns a `widget *`. If `p` were not a `widget *`, the compiler would complain about the assignment.

When allocating an array with `N` elements of type `T`, the appropriate type in the cast expression is still `T *`, but the argument to `malloc()` should be of the form `N * sizeof(T)`. Again, packaging this form as a macro, such as

```
#define MALLOC_ARRAY(number, type) \
    ((type *)malloc((number) * sizeof(type)))
```

reduces the chance of error in an allocation expression.

```
enum { N = 16 };
widget *p;

/* ... */

p = MALLOC_ARRAY(N, widget);    /* OK */
```

A small collection of macros can provide secure implementations for common uses for the standard memory allocation functions. The omission of a `REALLOC()` macro is intentional (see EXP39-C. Do not access a variable through a pointer of an incompatible type).

```
/* Allocates a single object using malloc() */
#define MALLOC(type) ((type *)malloc(sizeof(type)))

/* Allocates an array of objects using malloc() */
#define MALLOC_ARRAY(number, type) \
    ((type *)malloc((number) * sizeof(type)))

/*
 * Allocates a single object with a flexible
 * array member using malloc().
 */
#define MALLOC_FLEX(stype, number, etype) \
    ((stype *)malloc(sizeof(stype) \
    + (number) * sizeof(etype)))

/* Allocates an array of objects using calloc() */
#define CALLOC(number, type) \
    ((type *)calloc(number, sizeof(type)))

/* Reallocates an array of objects using realloc() */
#define REALLOC_ARRAY(pointer, number, type) \
    ((type *)realloc(pointer, (number) * sizeof(type)))

/*
 * Reallocates a single object with a flexible
 * array member using realloc().
 */
#define REALLOC_FLEX(pointer, stype, number, etype) \
    ((stype *)realloc(pointer, sizeof(stype) \
    + (number) * sizeof(etype)))
```

The following is an example:

```
enum month { Jan, Feb, /* ... */ };
typedef enum month month;

typedef struct date date;
struct date {
  unsigned char dd;
  month mm;
  unsigned yy;
};

typedef struct string string;
struct string {
  size_t length;
  char text[];
};

date *d, *week, *fortnight;
string *name;

d = MALLOC(date);
week = MALLOC_ARRAY(7, date);
name = MALLOC_FLEX(string, 16, char);
fortnight = CALLOC(14, date);
```

If one or more of the operands to the multiplication operations used in many of these macro definitions can be influenced by untrusted data, these operands should be checked for overflow before the macro is invoked (see INT32-C. Ensure that operations on signed integers do not result in overflow).

The use of type-generic function-like macros is an allowed exception (PRE00-C-EX4) to PRE00-C. Prefer inline or static functions to function-like macros.

Exceptions

MEM02-C-EX1: Do not immediately cast the results of `malloc()` for code that will be compiled using a C90-conforming compiler because it is possible for the cast to hide a more critical defect (see DCL31-C. Declare identifiers before using them for a code example that uses `malloc()` without first declaring it).

Risk Assessment

Failing to cast the result of a memory allocation function call into a pointer to the allocated type can result in inadvertent pointer conversions. Code that follows this recommendation will compile and execute equally well in C++.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MEM02-C	Low	Unlikely	Low	P3	L3

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**alloc-without-cast**	Partially checked
Axivion Bauhaus Suite	6.9.0	**CertC-MEM02**	Fully implemented
Compass/ROSE			Can detect some violations of this recommendation when checking EXP36-C. Do not cast pointers into more strictly aligned pointer types
ECLAIR	1.2	**CC2.MEM02**	Fully implemented
Parasoft C /C++test	10.4.2	**CERT_C-MEM02-a** **CERT_C-MEM02-b**	Assignment operator should have operands of compatible types Do not assign function return value to a variable of incompatible type
Polyspace Bug Finder	R2019a	CERT C: Rec. MEM02-C	Checks for wrong allocated object size for cast (rec. partially covered)
PRQA QA-C	9.5	**0695**	Fully implemented
RuleChecker	19.04	**alloc-without-cast**	Partially checked

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

SEI CERT C++ Coding Standard	VOID MEM02-CPP. Immediately cast the result of a memory allocation function call into a pointer to the allocated type

Bibliography

[Summit 2005]	Question 7.7 Question 7.7b